Interpreting Localized Computational Effects Using Operators of Higher Type

Interpreting localized computational effects using operators of higher type (extended abstract)

John Longley

Laboratory for Foundations of Computer Science
School of Informatics, University of Edinburgh
The King's Buildings, Mayfield Road
Edinburgh EH9 3JZ, UK

Abstract. We outline a general approach to providing intensional models for languages with computational effects, whereby the problem of interpreting a given effect reduces to that of finding an operator of higher type satisfying certain equations. Our treatment consolidates and generalizes an idea that is already implicit in the literature on game semantics. As an example, we work out our approach in detail for the case of fresh name generation, and discuss some particular models to which it applies.

1 Introduction

This paper explores a way in which computable operations of higher type can be useful in giving *denotational semantics* for programming languages. In broad terms, a denotational semantics for a language $\mathcal{L}$ consists of a mathematical model of the behaviour of programs in $\mathcal{L}$, given by assigning to each program P a "meaning" $[\![P]\!]$ within some mathematical structure $\mathcal{M}$ which can be defined and studied independently of the syntax of $\mathcal{L}$. Not only does this result in a rigorous mathematical definition of the programming language, but if $\mathcal{M}$ itself enjoys good mathematical properties, these can be used to reason about programs of $\mathcal{L}$. Furthermore, a particularly well-behaved model $\mathcal{M}$ might even inspire the design of a new and better programming language. For information on the mathematical aspects of denotational semantics, we recommend [7].

Much of the foundational work in denotational semantics focused initially on purely functional languages such as Plotkin's PCF [26]. The essence of such languages is that the behaviour of a program can be adequately modelled by a mathematical *function*, so that the language is amenable to a denotational description in terms of some well-understood mathematical class of functions, such as Scott's partial continuous functionals of higher type. However, whilst such purely functional settings are simple and mathematically appealing, virtually all real-world programming languages abound in "impure" features that break the simple-minded functional paradigm, such as exceptions, state, continuations, fresh name generation, input/output and nondeterminism. It is therefore natural to seek appropriate mathematical theories for modelling such computational *effects* (as they are generically known).

To date, there have broadly been two approaches to the denotational semantics of computational effects. The first, and more widely established, was pioneered by Moggi [20] in his investigation of the use of *monads* to model effects of various kinds. Here, a term $M : \sigma$, possibly involving some computational effect, is modelled by an element not of the usual object $[\![\sigma]\!]$, but of some richer domain $T[\![\sigma]\!]$, where T is a monadic functor chosen to match the effect in question. For instance, we may take $T(X) = X + E$ if the evaluation of M may result in an exception drawn from the set E, or $T(X) = (S \times X)^S$ if the evaluation may have a side-effect on some state of type S, and so on. In this way, an essentially "functional" treatment of programs is maintained, at the cost of complicating the types of the functions involved: typically, a program of type $\sigma \to \tau$ will be modelled by a function of type $[\![\sigma]\!] \to T[\![\tau]\!]$ rather than $[\![\sigma]\!] \to [\![\tau]\!]$. The monadic approach has had a wide influence and some notable successes — in particular, it underpins the model of effects employed in the Haskell programming language [24]. More recently, a closely related approach has been developed by Plotkin and Power [27], emphasizing the primacy of Lawvere-style *algebraic theories* rather than monads — this promises, among other things, a more satisfactory account of how different computational effects may be combined in a principled way.

By contrast, the second main approach (advocated explicitly in [1]) seeks to model programs with effects not by functions of more complicated types, but by more fine-grained, *intensional* semantic objects than ordinary mathematical functions — typically algorithms, strategies or even programs of some kind. Such intensional notions of "computable operation" occasionally featured in the earlier literature on higher types (*e.g.* the non-extensional type structures HRO and ICF derived from Kleene's first and second models respectively [30]); in the computer science literature, an important early example was the *sequential algorithm* model of Berry and Curien [8]. A wealth of further interesting models were subsequently introduced by the literature on *game semantics* [5, 3, 4, 12], where numerous full abstraction results were obtained for languages with control features, state, and non-determinism in various combinations (see [11] for a survey, and [19] for a useful taxonomy of models). Whilst these intensional models may appear unfamiliar at first, experience shows that many of them lead to beautiful mathematical structures, carry a persuasive intuition, and (in the author's view) provide good candidates for notions of higher type computability in the spirit of [16].

Although numerous examples of intensional models for languages with effects have now been collected, they have so far not conformed to much of a general pattern. Our purpose in the present article is to outline a somewhat uniform approach to the interpretation of computational effects in intensional models by means of operators of higher type. Particular instances of our approach may be discerned in the existing literature on game semantics, but as far as we know, the general idea has not hitherto been spelt out as a uniformly applicable method. Our presentation will, moreover, be at a level of generality which renders the ideas applicable to other intensional models besides game models.

The basic idea is as follows. Each kind of computational effect is typically associated with some characteristic syntactic operators: *e.g.* `raise` and `handle` in the case of exceptions; `read` and `write` in the case of store cells; `new` and `eq` in the case of the generation of fresh names (with equality testing), and so on. For the sake of discussion let us work with the example of fresh name generation, though the same basic strategy will clearly make sense for many other effects. In the context of a higher order (say call-by-value) language, we may naturally ascribe *types* to the relevant operators: *e.g.* in the spirit of the *ν-calculus* of Pitts and Stark [25], we have operations $\texttt{new} : \texttt{unit} \to \texttt{name}$ and $\texttt{eq} : \texttt{name} * \texttt{name} \to \texttt{bool}$. However, rather than attempting to model what these operators actually do, let us choose to regard them simply as *variables* that may appear in a term, with the same formal status as ordinary program variables. Thus, no special technology is needed at this stage to model terms involving such operators. However, in an intensional model, the denotation of such a term M may typically record information concerning when, and in what order, the characteristic operators are invoked, and how the results affect the subsequent computation. This means that (in good cases) the denotation of M (or equivalently of its closure $\overline{M} = \lambda \texttt{new}, \texttt{eq}.\, M$) will in principle contain enough information to determine how M *would* behave if genuine implementations of the appropriate operators were supplied.[1] Furthermore, in many cases, one can find within the model itself a higher order operator Φ which transforms the denotation of $\overline{M}$ to an element modelling the desired actual behaviour of M. One may informally think of Φ as modelling the behaviour of the program $\lambda F.\, F\ \texttt{New}\ \texttt{Eq}$, where `New`, `Eq` are actual implementations of the relevant operators; note that such a Φ may exist even though `New`, `Eq` themselves have no standalone interpretation in the model.

This naturally raises the question: what properties must an operator Φ satisfy in order to give rise to a correct semantics for fresh name generation? Our "reference semantics" for freshness will presumably be derived from our operational understanding of `New` and `Eq`, but we would also like a denotational (*e.g.* an equational) condition on Φ within the model which is sufficient and (ideally) necessary for the soundness of our interpretation. An operator Φ satisfying this condition may then be dubbed a *freshness operator*.

Having arrived at this general definition, it is then natural to ask which particular intensional models possess a freshness operator. We may think of this property as capturing something interesting about the innate computational power of a model (somewhat akin, say, to the property of having a fixed point operator or a modulus of continuity operator of some sort — see *e.g.* [30]), as well as its potential usefulness in denotational semantics. Moreover, by formulating this notion uniformly for a class of models, we facilitate the task of comparing and classifying models, thus contributing to the author's project of mapping out the landscape of computability notions [16, 17].

[1] In the case of store cells with `read` and `write` operations, this is very much how the interpretation *e.g.* in [3] works; this is perhaps the clearest manifestation in the existing literature of our basic idea.

Typically, our approach will work at its best for uses of computational effects that are *localized* to some block of code M (*cf.* [15]). A program that makes global use of some effect will be modelled as the denotation of an open term with free variables for the characteristic operations; the operation of localizing this effect then corresponds to abstracting over these variables and applying the appropriate operator Φ. Whilst this in principle allows us to interpret both complete "closed" programs and "open" fragments thereof, a common situation will be that our interpretation is fully abstract for closed programs, but very far from this for open ones.[2] A natural methodology is therefore to focus initially on the well-behaved situation for closed programs, and then to consider how the benefits of our interpretation might be extended to open programs. We will return to this issue in Section 4.

In the present article, we make a modest start on demonstrating the viability of our programme, focusing in particular on name generation. This seems an interesting example to consider for two reasons. Firstly, it cannot be straightforwardly modelled by monads on familiar categories of domains. This led Moggi originally to suggest using a monad on a *functor category* [21], an approach which has subsequently proved rather difficult to combine with other language features [28]. (In a different guise, functor categories are also an important ingredient in the Plotkin-Power approach to name generation — see [27, 29].) Secondly, virtually all other efforts to model name generation have, in some way, made essential use of another idea: that of a set of names acted on by a permutation group in order to make them "indistinguishable" [28, 13, 2, 31]. These approaches have achieved significant success, *e.g.* in terms of full abstraction results; however, we believe it is also interesting to explore how much can be achieved without resorting to the machinery of either functor categories or permutation actions. In particular, our approach shows that many models of computation that have proved to be of interest for other reasons (*e.g.* game models) already have what it takes to model name generation without any specialized additional technology.

The rest of the paper is structured as follows. In Section 2 we sketch a general framework (based on Moggi's notion of a λ_c-*model*) within which the general idea works out smoothly. In Section 3, as a concrete example, we consider the case of fresh name generation in some detail, including the definition of freshness operators and some technical results validating this definition. In Section 4 we survey some particular examples of models in which freshness operators are available, and in Section 5 we mention some avenues for further investigation.

The ideas in this paper arose rather naturally in the course of an attempt to design a programming language based around the structure available in a certain game model. For an account of this work in progress, see [18].

I am grateful to the CiE organizers for the invitation to present this material, to the reviewers for their helpful comments, and to Ian Stark and Nicholas Wolverson for valuable discussions. The research was supported by EPSRC Grant GR/T08791: "A programming language based on game semantics".

[2] In the case of store cells, this is related to the problem of "bad variables" — see [3].

2 The general framework

Although we will not be using monads themselves to model effects, we are indebted to the monadic tradition for a general notion of (intensional) model that is suited to our purposes. Because of the special role played by *values* (*i.e.* fully evaluated expressions) in programming languages with effects, it is convenient to frame our ideas in a call-by-value setting, and here a very suitable notion of model is provided by Moggi's work on *computational* λ*-calculus* [20]. The definition is most compactly presented in categorical terms. Formally, a λ_c*-model* is a category $\mathcal{C}$ with finite products, equipped with a strong monad (T, η, μ, t), such that for any $A, B \in \mathcal{C}$ the exponential TB^A exists (we henceforth denote TB^A by $A \Rightarrow B$).[3] For convenience, we assume our λ_c-models come equipped with objects $1, 2$ representing unit and boolean types. We abbreviate $f : 1 \to A$ to $f \in A$.

The intuition is that whereas an object A may serve for modelling *values* of some type, the corresponding object TA will model more general expressions of this type whose evaluation may involve some effect. (For a detailed explanation of why strong monads are an appropriate choice of structure here, we refer the reader to [22].) In Moggi's work, one considers a range of different monads to capture different computational effects. By contrast, here we will be concerned almost exclusively with *lifting* monads representing potentially non-terminating computations — the interest for us lies in varying the base category $\mathcal{C}$ to capture different "levels of intensionality". However, it is worth remarking that computability models that are *too* finely intensional (such as those based on Kleene's models K_1 and K_2) fail even to be λ_c-models, and it is not clear whether our approach can be made to work at all in these settings.

Rather than writing lengthy categorical expressions, we use shall use a familiar lambda-calculus notation for denoting morphisms of $\mathcal{C}$ as in [14]; the precise intention in any given instance will be clear from the types involved. We shall supplement this with some meta-notation borrowed from [20]: we write $[e]$ for the inclusion of $e : A$ into TA via η_A, and $\mathsf{let}\ x = e\ \mathsf{in}\ e'$ for the "Kleisli composition" of e and e'.[4] The essential point about the latter is that it captures the call-by-value discipline of forcing the evaluation of e whether or not x appears in e'. We write $\mathsf{let}\ \boldsymbol{a} = e^r\ \mathsf{in}\ e'$ to abbreviate $\mathsf{let}\ a_1 = e\ \mathsf{in}\ \cdots\ \mathsf{let}\ a_r = e\ \mathsf{in}\ e'$.

We also introduce some syntactic machinery intended to embody the general notion of a "programming language interpretable in $\mathcal{C}$". We represent the syntax of such a language as a category *à la* Lawvere, with composition corresponding to syntactic substitution. Formally, an *object language* $\mathcal{L}$ consists of:

[3] The "mono requirement" mentioned in [20] is not needed for our purposes.

[4] More precisely, if e is a meta-expression of type TA involving metavariables $y_i : C_i$, and e' a meta-expression of type TB involving metavariables $y_i : C_i$ and $x : A$, then $\mathsf{let}\ x = e\ \mathsf{in}\ e'$ denotes the morphism

$$\Pi C_i \xrightarrow{\langle id, e\rangle} \Pi C_i \times TA \xrightarrow{t_{\Pi C_i, A}} T(\Pi C_i \times A) \xrightarrow{Te'} T(TB) \xrightarrow{\mu_B} TB$$

For a more formal treatment, see [22].

– a collection of *types* σ, τ, equipped with binary operations $*, \to$ and including for convenience the types $\mathtt{unit}$ and $\mathtt{bool}$;
– a category with finite products whose objects are finite tuples of types $(\sigma_1, \ldots, \sigma_n)$ (the product of two objects being their concatenation as tuples).

Morphisms $(\sigma_1, \ldots, \sigma_n) \to (\tau)$ should be thought of as equivalence classes of terms-in-context $x_1 : \sigma_1, \ldots, x_n : \sigma_n \vdash M : \tau$ modulo renaming of variables. We shall generally use terms-in-context to denote morphisms of $\mathcal{L}$ and blur the distinction between the two notions. We shall also require that there are constant terms $\langle\,\rangle : \mathtt{unit}$ and $\mathtt{tt}, \mathtt{ff} : \mathtt{bool}$, and for each σ, τ there exist pairing and application terms $x : \sigma, y : \tau \vdash \langle x, y\rangle : \sigma * \tau$ and $f : \sigma \to \tau, x : \sigma \vdash f \bullet x : \tau$. (We warn the reader against confusion between the object language syntax and the meta-notation conventions introduced above.)

A *programming language* will be an object language $\mathcal{L}$ endowed with an operational semantics which includes a notion of "symbolic evaluation" for open terms as well as the usual notion of evaluation for closed terms. Formally, we shall require:

– for each Γ, τ, a reflexive-transitive *evaluation* relation $\twoheadrightarrow$ (more properly written $\Gamma \vdash - \twoheadrightarrow - : \tau$) on terms $\Gamma \vdash M : \tau$;
– for each Γ, σ, a set of terms of $\mathcal{L}$ in context $\Gamma, - : \sigma$ designated as *evaluation contexts* $E[-]$ (where '$-$' is a distinguished free variable).

In typical cases, these will satisfy further properties, *e.g.*:

– if $M \twoheadrightarrow N$ and $E[-]$ is an evaluation context then $E[M] \twoheadrightarrow E[N]$;
– for any Γ, τ, Γ', the term $\Gamma, -, \Gamma' : \tau \vdash - : \tau$ is an evaluation context;
– if $E[-], E'[-]$ are appropriately typed evaluation contexts then so is their evident composition $E'[E[-]]$;
– for any M and x, the terms $- \bullet M$, $x \bullet -$, $\langle -, M\rangle$ and $\langle x, -\rangle$ (in any suitable context $\Gamma, - : \tau$) are evaluation contexts.

A language satisfying these properties will be called *standard*. Surprisingly, however, none of these properties will be required for our main results.

An *interpretation of types of* $\mathcal{L}$ *in a* λ_c*-model* $\mathcal{C}$ is a mapping $[\![-]\!]$ from types of $\mathcal{L}$ to objects of $\mathcal{C}$ satisfying the expected properties: $[\![\,\mathtt{unit}\,]\!] = 1$, $[\![\,\mathtt{bool}\,]\!] = 2$, $[\![\,\mathtt{nat}\,]\!] = N$, $[\![\,\sigma * \tau\,]\!] = [\![\,\sigma\,]\!] \times [\![\,\tau\,]\!]$, $[\![\,\sigma \to \tau\,]\!] = [\![\,\sigma\,]\!] \Rightarrow [\![\,\tau\,]\!]$. By convention, if $\Gamma = x_1 : \sigma_1, \ldots, x_r : \sigma_r$ we write $[\![\,T\Gamma\,]\!]$ for the object $T[\![\,\sigma_1\,]\!] \times \cdots \times T[\![\,\sigma_r\,]\!]$. Note the appearance of T here, which contrasts with the modelling of contexts *e.g.* in [20]. This reflects the fact that the role of variables here is purely to allow us to talk about the compositional structure of terms rather than to model any kind of object-level variable binding.

Relative to such a mapping, an *interpretation* $[\![-]\!]$ *of* $\mathcal{L}$ maps each term $\Gamma \vdash M : \tau$ of $\mathcal{L}$ to a morphism $[\![\,M\,]\!]_\Gamma : [\![\,T\Gamma\,]\!] \to T[\![\,\tau\,]\!]$, in such a way that

– variables, constants, application and pairing receive the expected (left-strict) interpretation, and the evident weakening and contraction properties hold;

– $[\![-]\!]$ is *compositional*: that is, if $\Delta_i \vdash N_i : \sigma_i$ for each i (with the Δ_i disjoint) and $\Gamma = x_1 : \sigma_1, \ldots, x_r : \sigma_r \vdash M : \tau$, then

$$[\![M[\boldsymbol{N}/\boldsymbol{x}]]\!]_{\boldsymbol{\Delta}} \;=\; [\![M]\!]_\Gamma \circ ([\![N_1]\!]_{\Delta_1} \times \ldots \times [\![N_r]\!]_{\Delta_r})$$

– if $\Gamma, - : \sigma \vdash E[-] : \tau$ is an evaluation context then $[\![E[-]]\!]_{\Gamma,-:\sigma}$ is strict in '$-$', that is:

$$[\![E[-]]\!](\boldsymbol{x}, y) \;=\; \mathsf{let}\ z = y\ \mathsf{in} [\![E]\!](\boldsymbol{x}, [z])$$

Clearly, a global interpretation gives rise to a functor $\mathcal{L} \to \mathcal{C}$ which preserves finite products. Note, however, that products in $\mathcal{L}$ do *not* correspond to object-level product types, the difference being that between $TA \times TB$ and $T(A \times B)$.

An interpretation $[\![-]\!]$ is called *sound* if $\Gamma \vdash M \twoheadrightarrow N$ implies $[\![M]\!]_\Gamma = [\![N]\!]_\Gamma$. We also say $[\![-]\!]$ is *sound for* a class $\mathcal{T}$ of $\mathcal{L}$-terms if this property holds whenever $M \in \mathcal{T}$.

3 Fresh name generation

As our main example, we now work out our approach in some detail for the case of fresh name generation in the spirit of the ν-calculus. We first present our operational understanding of name generation by describing how any programming language $\mathcal{L}$ may be extended to a language $\mathcal{L}^+$ with (localized) name generators. We then investigate the conditions under which an interpretation for $\mathcal{L}$ may be extended to one of $\mathcal{L}^+$.

First, let us assume for convenience that our original object language $\mathcal{L}$ comes already equipped with an infinite supply of *name types*, ranged over by ν. (These types need play no active role in $\mathcal{L}$ beyond what is implied by the fact of being a finite-product category.) We may then freely extend $\mathcal{L}$ (as a finite-product category) to a language $\mathcal{L}^+$ by means of the following term formation rule:

$$\frac{\Gamma,\ new : \mathtt{unit} \to \nu,\ eq : \nu * \nu \to \mathtt{bool},\ a_1 : \nu, \ldots, a_r : \nu \ \vdash\ M : \tau}{\Gamma \ \vdash\ \mathtt{gen}_\nu, new, eq, (a_1, \ldots, a_r)\ \mathtt{in}\ M \quad : \tau} \quad \nu \notin \Gamma, \tau$$

where ν ranges over name types and $new, eq, \boldsymbol{a}$ may be any variables.[5] If $\mathcal{G} \subseteq \mathcal{T}$ is any set of types, we also obtain a restricted language $\mathcal{L}^+_{\mathcal{G}}$ by requiring that $\tau \in \mathcal{G}$ in the above rule. This construct allows us to introduce a localized name generator with characteristic operations represented by *new* and *eq*, whose generated names are prevented by the type system from being confused with names arising from other generators. (The side-condition also ensures that the generated names cannot leak out of their scope.) This kind of potentially nested block structure sets the pattern for our general approach to localized effects; in the present context it also models the situation *e.g.* in Standard ML, where a

[5] We regard $\nu, new, eq, \boldsymbol{a}$ as being *bound* by this construction, and should also require in the above rule that they do not occur bound within M itself. Technically, we view α-equivalent expressions as defining the same term in $\mathcal{L}^+$, so that for the purpose of substitution and evaluation we may freely apply α-conversion as necessary.

local datatype declaration implicitly introduces a localized name generator for the corresponding `ref` type. The a_i play the role of names that have already been created using the generator in question, and which may feature in M.

We let $E^*[-]$ range over $\mathcal{L}^+$-substitution instances of evaluation contexts in $\mathcal{L}$, and let $G[-]$ range over *gen-contexts*: that is, compositions of zero or more contexts of the form $\mathtt{gen}\ new, eq, (\boldsymbol{a})\ \mathtt{in}\ -$. As evaluation contexts of $\mathcal{L}^+$ or $\mathcal{L}^+_{\mathcal{G}}$, we take all contexts of the form $G[E^*[-]]$ (note that these are *not* closed under composition), and let $F[-]$ range over these. As an operational semantics, we let $\twoheadrightarrow^+$ be the evaluation relation generated by the following:

- if $M \twoheadrightarrow N$ in $\mathcal{L}$ and $\boldsymbol{P}$ is a list of $\mathcal{L}^+$-terms, then $G[M[\boldsymbol{P}/\boldsymbol{x}]] \twoheadrightarrow^+ G[N[\boldsymbol{P}/\boldsymbol{x}]]$;
- $\mathtt{gen}_\nu\ new, eq, (\boldsymbol{a})\ \mathtt{in}\ F[new \bullet \langle\,\rangle] \twoheadrightarrow^+ \mathtt{gen}_\nu\ new, eq, (\boldsymbol{a}, b)\ \mathtt{in}\ F[a']$ (a' fresh);
- $\mathtt{gen}_\nu\ new, eq, (\boldsymbol{a})\ \mathtt{in}\ F[eq \bullet \langle a_i, a_i\rangle] \twoheadrightarrow^+ \mathtt{gen}_\nu\ new, eq, (\boldsymbol{a})\ \mathtt{in}\ F[\mathtt{tt}]$;
- $\mathtt{gen}_\nu\ new, eq, (\boldsymbol{a})\ \mathtt{in}\ F[eq \bullet \langle a_i, a_j\rangle] \twoheadrightarrow^+ \mathtt{gen}_\nu\ new, eq, (\boldsymbol{a})\ \mathtt{in}\ F[\mathtt{ff}]$ ($i \neq j$);
- $G[E^*[\mathtt{gen}_\nu\ new, eq, (\boldsymbol{a})\ \mathtt{in}\ M]] \twoheadrightarrow^+ G[\mathtt{gen}_\nu\ new, eq, (\boldsymbol{a})\ \mathtt{in}\ E^*[M]]$;
- $G[\mathtt{gen}_\nu\ new, eq, (\boldsymbol{a})\ \mathtt{in}\ M] \twoheadrightarrow^+ G[M]$ if $new, eq, \boldsymbol{a}$ do not appear in M.

We will take the definition of $\twoheadrightarrow^+$ as our reference semantics for fresh name generation, and ask when an interpretation in a λ_c-model accords with this. The last of the above rules is a "garbage collection rule" — the definition of $\twoheadrightarrow^+$ admits some non-determinism regarding exactly when this rule is to be applied, but this does not matter for our purposes, since Propositions 1 and 2 below will apply *a fortiori* to any reduction strategy included in $\twoheadrightarrow^+$. Note also that if $\mathcal{L}$ is standard, the following are always evaluation contexts:

$$new \bullet - \qquad eq \bullet \langle -, M\rangle \qquad eq \bullet \langle a_i, -\rangle$$

and this enables us to make progress with the evaluation of programs such as

$$\mathtt{gen}_\nu\ new, eq, ()\ \mathtt{in}\ eq \bullet \langle new \bullet \langle\,\rangle,\ (\mathtt{fn}\ \mathtt{x} => new \bullet \langle\,\rangle) \bullet \langle\,\rangle\rangle$$

Next, suppose we are given an interpretation $[\![-]\!]$ of $\mathcal{L}$ in $\mathcal{C}$, and assume for simplicity that $[\![\,\nu\,]\!]$ is the same object A_{name} for all name types ν. Let $A_{new} = 1 \Rightarrow A_{name}$ and $A_{eq} = A_{name} \times A_{name} \Rightarrow 2$. Suppose moreover that for each $\tau \in \mathcal{G}$ we are given some operator $\Phi_\tau \in (A_{new} \Rightarrow A_{eq} \Rightarrow [\![\,\tau\,]\!]) \Rightarrow [\![\,\tau\,]\!]$, and let Φ denote the indexed family $\{\Phi_\tau \mid \tau \in \mathcal{G}\}$. Relative to Φ, we may define an interpretation $[\![-]\!]^\Phi$ for terms of $\mathcal{L}^+_{\mathcal{G}}$ as follows:

- $[\![\,M\,]\!]^\Phi_\Gamma = [\![\,M\,]\!]_\Gamma$ if $\Gamma \vdash M : \tau$ in $\mathcal{L}$;
- $[\![\,\mathtt{gen}_\nu\ new, eq, (a_1, \ldots, a_r)\ \mathtt{in}\ M\,]\!]^\Phi_\Gamma$ is taken to be

$$\lambda \boldsymbol{x} : [\![\,T\Gamma\,]\!].\ \Phi_\tau\,(\lambda \mathsf{new}, \mathsf{eq}.\ \mathsf{let}\ \boldsymbol{a} = (\mathsf{new}\,\langle\,\rangle)^r\ \mathsf{in}\ [\![\,M\,]\!]^\Phi_{\Gamma'}\,(\boldsymbol{x}, [\mathsf{new}], [\mathsf{eq}], [a_1], \cdots, [a_r]))$$

  where $\Gamma' = \Gamma,\ new : \mathtt{unit} \to \nu,\ eq : \nu * \nu \to \mathtt{bool},\ a_1 : \nu, \ldots, a_r : \nu$;
- $[\![-]\!]^\Phi$ extends to arbitrary terms of $\mathcal{L}^+_{\mathcal{G}}$ via compositionality.

Under what conditions is this interpretation a reasonable one? We propose the following semantic definition; note that only *equations* between higher type operators are involved. For readability, we take a few small liberties in our meta-notation, *e.g.* writing $f\ \mathsf{new}\,\mathsf{eq}\,\langle b, \boldsymbol{a}\rangle$ in place of $\mathsf{let}\ f' = f\ \mathsf{in}\ f'\,\mathsf{new}\,\mathsf{eq}\,\langle b, \boldsymbol{a}\rangle$.

Definition 1. *A* freshness operator *for a type τ is an operator $\Phi_\tau \in (A_{new} \Rightarrow A_{eq} \Rightarrow [\![\tau]\!]) \Rightarrow [\![\tau]\!]$ satisfying the following equations.*

1. *If $A_g = T[\![\tau]\!]$ then*

$$\lambda g : A_g.\ \Phi_\tau(\lambda \mathsf{new}, \mathsf{eq}.\,\mathsf{let}\ \boldsymbol{a} = (\mathsf{new}\,\langle\,\rangle)^r\ \mathsf{in}\ g) \quad = \quad \lambda g : A_g.\ g$$

2. *For all r and all $i \leq r$, if $A_f = T(A_{new} \Rightarrow A_{eq} \Rightarrow (2 \times A_{name}^r) \Rightarrow [\![\tau]\!])$ then*

$$\begin{array}{c}\lambda f : A_f.\ \Phi_\tau(\lambda \mathsf{new}, \mathsf{eq}.\,\mathsf{let}\ \boldsymbol{a} = (\mathsf{new}\,\langle\,\rangle)^r\,\mathsf{in}\,\mathsf{let}\ b = \mathsf{eq}\,(a_i, a_i)\,\mathsf{in}\ f\,\mathsf{new}\,\mathsf{eq}\,\langle b, \boldsymbol{a}\rangle) \\ = \quad \lambda f : A_f.\ \Phi_\tau(\lambda \mathsf{new}, \mathsf{eq}.\,\mathsf{let}\ \boldsymbol{a} = (\mathsf{new}\,\langle\,\rangle)^r\ \mathsf{in}\ f\,\mathsf{new}\,\mathsf{eq}\,\langle \mathsf{tt}, \boldsymbol{a}\rangle)\end{array}$$

3. *For all r and all $i, j \leq r$ with $i \neq j$, if A_f is as above then*

$$\begin{array}{c}\lambda f : A_f.\ \Phi_\tau(\lambda \mathsf{new}, \mathsf{eq}.\,\mathsf{let}\ \boldsymbol{a} = (\mathsf{new}\,\langle\,\rangle)^r\,\mathsf{in}\,\mathsf{let}\ b = \mathsf{eq}\,(a_i, a_j)\,\mathsf{in}\ f\,\mathsf{new}\,\mathsf{eq}\,\langle b, \boldsymbol{a}\rangle) \\ = \quad \lambda f : A_f.\ \Phi_\tau(\lambda \mathsf{new}, \mathsf{eq}.\,\mathsf{let}\ \boldsymbol{a} = (\mathsf{new}\,\langle\,\rangle)^r\ \mathsf{in}\ f\,\mathsf{new}\,\mathsf{eq}\,\langle \mathsf{ff}, \boldsymbol{a}\rangle)\end{array}$$

Although a certain amount can be achieved even with a single freshness operator, more can be done with a family of such operators that fit well together. A family $\Phi = \{\Phi_\tau \mid \tau \in \mathcal{G}\}$ of freshness operators is called *coherent* if for any $\sigma, \tau \in \mathcal{G}$, writing $A_c = T[\![\sigma \to \tau]\!]$ and $A_p = T(A_{new} \Rightarrow A_{eq} \Rightarrow [\![\sigma]\!])$ we have

$$\begin{array}{c}\lambda c : A_c.\ \lambda p : A_p.\ \Phi_\tau(\lambda \mathsf{new}, \mathsf{eq}.\,\mathsf{let}\ x = p\ \mathsf{new}\ \mathsf{eq}\ \mathsf{in}\ c\,x) \\ = \quad \lambda c : A_c.\ \lambda p : A_p.\ \mathsf{let}\ x = \Phi_\sigma p\ \mathsf{in}\ c\,x\end{array}$$

(In particular, the Φ_τ form a natural transformation $(A_{new} \Rightarrow A_{eq} \Rightarrow -) \to -$ considered as functors on the relevant portion of the Kleisli category $\mathcal{C}_T$.)

We now state a series of results which collectively validate the above definitions, and also confirm the appropriateness of our framework as a whole. (We omit the rather straightforward proofs.) We henceforth assume a fixed sound interpretation $[\![-]\!]$ of $\mathcal{L}$ in $\mathcal{C}$. By a *primary term* we mean an $\mathcal{L}^+$-term of the form $G[M]$ where $M \in \mathcal{L}$.

Proposition 1. *(i) If Φ_τ is a freshness operator for τ, then $[\![-]\!]^\Phi$ as defined above constitutes an interpretation of $\mathcal{L}^+_{\{\tau\}}$ which is sound for primary terms of type τ.*

(ii) Moreover, if Φ is a coherent family of freshness operators for $\mathcal{G}$, then $[\![-]\!]^\Phi$ is a sound interpretation of $\mathcal{L}^+_{\mathcal{G}}$.

As an immediate consequence, we have the following:

Proposition 2. *(i) Suppose Φ is a freshness operator for τ, M is a closed primary term of type τ, and N a closed $\mathcal{L}$-term of type τ. Then $\emptyset \vdash M \twoheadrightarrow^+ N : \tau$, implies $[\![M]\!]^\Phi = [\![N]\!]$.*

(ii) Suppose Φ is a coherent family of freshness operators for $\mathcal{G}$, M is an arbitrary closed $\mathcal{L}^+_{\mathcal{G}}$-term of type $\tau \in \mathcal{G}$, and N a closed $\mathcal{L}$-term of type σ. Then $\emptyset \vdash E[M] \twoheadrightarrow^+ N$ implies $[\![E[M]]\!]^\Phi = [\![N]\!]$.

The moral of Proposition 2 is roughly as follows. Typically, we will be interested in languages $\mathcal{L}$ with a designated class of syntactic values V, and a class of *ground types* γ with the property that a value V of ground type cannot contain any variables new, eq, a_i with types as above. Thinking of N in Proposition 2 as ranging over values, the proposition implies (in typical cases) that to obtain a sound interpretation for programs containing only `gen` expressions of ground type, the existence of the corresponding freshness operators is sufficient; however, to give an interpretation programs involving non-ground type localizations which correctly accounts for ground type observations on them, coherence is required. This phenomenon is not special to the case of name generation, but appears to be typical of our approach.

The converse half of computational adequacy requires stronger hypotheses, such as a "syntactic continuity" property, though we will not go into the details here. For our present purposes, a more interesting kind of converse is the following, which can be seen as validating our definition of freshness operator:

Proposition 3. *(i) Suppose Φ_τ is an operator such that for every programming language $\mathcal{L}'$ with a sound interpretation $[\![-]\!]'$ in $\mathcal{C}$ (agreeing on types with $[\![-]\!]$), the interpretation $[\![-]\!]'^{\Phi_\tau}$ of $\mathcal{L}'^{+}_{\{\tau\}}$ is sound for primary terms of type τ. Then Φ is a freshness operator for $[\![\tau]\!]$.*

(ii) Suppose Φ is a family of operators over $\mathcal{G}$ such that for every programming language $\mathcal{L}'$ with a sound interpretation $[\![-]\!]'$ in $\mathcal{C}$, $[\![-]\!]'^{\Phi}$ is a sound interpretation of $\mathcal{L}'^{+}_{\mathcal{G}}$. Then Φ is a coherent family of freshness operators.

Note also that in the above setting, the freshness operators are themselves syntactically definable in $\mathcal{L}^+$, since $\Phi_\tau = [\![\, \mathtt{gen}_\nu\ new, eq, ()\ \mathtt{in}\ (-\bullet new \bullet eq)\,]\!]^{\Phi}$. In fact, it seems reasonable to suppose that for *any* natural interpretation of $\mathcal{L}^+$, the operator defined in this way will be a freshness operator. Thus, if we are seeking to interpret a language with name generation in an intensional model, little or no useful generality appears to be lost by assuming the existence of freshness operators.

4 Models

We now briefly review what we know concerning particular λ_c-models that possess freshness operators. Firstly, any of the known game models that suffice for modelling local store of integer type (see *e.g.* [5, 3, 6]) will also yield a model for fresh name generation according to our scheme, for the simple reason that (taking names to be just integers) a freshness operator may be readily implemented using a local integer store cell. (Note that the game models in question may be transformed into suitable λ_c-models by means of a standard construction [4].)

Some idea of the landscape may be gained by considering a few particular game models that are relatively simple to construct. We content ourselves here with a bare sketch of the relevant points, referring to the literature for further details. Let $\mathcal{G}$ denote the basic game model introduced by Lamarche (see [10]): here, a game G consists of sets O_G, P_G of *opponent* and *player* moves respectively,

together with a non-empty prefix-closed set L_G of *legal positions* of the form $o_1p_1 \ldots o_np_n$ $(n \geq 0)$ or $o_1p_1 \ldots o_n$ $(n \geq 1)$, where $o_i \in O_G$, $p_i \in P_G$. Such games (with suitable morphisms) form a symmetric monoidal closed category, on which one may consider several different linear exponentials '!' embodying different notions of "reusability". From any of these exponentials we may obtain a category $\mathcal{G}_!$ with the same objects as $\mathcal{G}$, in which morphisms $G \to H$ are simply morphisms $!G \to H$ of $\mathcal{G}$. This gives a cartesian-closed category with a lifting monad, and hence a suitable λ_c-model.

Some exponentials of particular interest are the following (*cf.* [19]):

- The "Lamarche exponential" $!_1$. Here moves in $!_1G$ are certain finite subtrees of L_G, and a play in $!_1G$ consists of an "exploration" of L_G in which one new position $s \in L_G$ is added to the subtree at each stage. From the corresponding category $\mathcal{G}_{!_1}$ one recovers essentially the world of sequential algorithms [10]. However, this does *not* yield a model for either ground type store or freshness, essentially because repetitions of earlier moves are not accounted for in $!_1G$.
- The (more powerful) "Hyland exponential" $!_2$ of [12]. Here $!_2G$ essentially consists of ω copies of G side by side, with the stipulation that one cannot play a move in the $(i+1)$th copy unless one has already played in the ith copy. The category $\mathcal{G}_{!_2}$ gives a good model for ground type store and more besides [32], and in particular has a coherent family of freshness operators.
- A still more powerful exponential $!_3$ may be defined, where plays in $!_3G$ explore trees of *justified sequences* of moves in G. This essentially coincides with the exponential given in [5], except that we do not impose a visibility condition on our plays. Again, the corresponding model supports ground-type store and freshness operators.

We are also aware of one model which is *not* a game model, and which supports freshness operators but not local store. This provides an encouraging sign that our general approach is applicable beyond the class of models that motivated it. The model in question is based on a "resource-sensitive" model for linear logic, in which *multisets* are used to keep track of the number of times some argument is invoked in a computation, but without imposing a temporal order on these invocations as the game models do. Specifically, we have in mind the category **MRel**, whose objects are sets and whose morphisms $f : S \to T$ are relations $f \subseteq \mathcal{M}_f(S) \times T$, where $\mathcal{M}_f(S)$ is the set of finite multisets over S. (An explicit description of this category and its cartesian closed structure is given in [9].) We may also endow **MRel** with a (rather crude) lifting monad which simply adds to each set a new token $*$ signalling "definedness", and again apply the construction of [4] to obtain a λ_c-model. Within this model, it is possible to "probe" an operation $p : A_{new} \Rightarrow A_{eq} \Rightarrow X$ in order to discover what it does when all invocations of A_{new} yield different answers. Using this idea, we obtain a coherent family of freshness operators within the model.

We now return to the question of full abstraction mentioned in the Introduction. According to the setup of Section 3, if the interpretation $[\![-]\!]$ of $\mathcal{L}$ in $\mathcal{C}$

satisfies full abstraction and definability, then so will the resulting interpretation $[\![-]\!]^{\Phi}$ of $\mathcal{L}^+$. However, this relies on the fact that, in $\mathcal{L}^+$, the characteristic operators *new* and *eq* are just ordinary variables, whereas in more realistic languages they will be hard-wired in as language primitives, as in the original ν-calculus [25]. In the latter case, we can still get a semantic interpretation

$$M : \sigma \quad \mapsto \quad [\![M]\!] \in A_{new} \Rightarrow A_{eq} \Rightarrow [\![\sigma]\!]$$

by treating *new* and *eq* as free variables, though this will (in game models, for instance) not even validate such simple observational equivalences as

$$\texttt{let (x,y)=(new(),new()) in } M \simeq \texttt{let (y,x)=(new(),new()) in } M$$

To do better than this, an alternative (and still compositional) interpretation

$$M : \sigma \quad \mapsto \quad [\![M]\!]^{\dagger} \in (A_{new} \Rightarrow A_{eq} \Rightarrow T[\![\sigma]\!] \Rightarrow 1) \Rightarrow 1$$

may cheaply be defined from $[\![-]\!]$ as follows:

$$[\![M]\!]^{\dagger} \;=\; \lambda P.\, \Phi\,(\lambda \mathsf{new}, \mathsf{eq}.\, (P\,\mathsf{new}\,\mathsf{eq})([\![M]\!]\,\mathsf{new}\,\mathsf{eq}))$$

In typical models, $[\![-]\!]^{\dagger}$ will validate simple equivalences such as the one above, at least at low types. Whether a fully abstract semantics for (extensions of) the ν-calculus can be given along these lines is an interesting outstanding question.

5 Further work and conclusions

The next step in our programme is to carry out a similar analysis for other computational effects. We have informally verified that a similar story can be told for exceptions and for (ground or higher type) local store. In the case of exceptions, two options are available. The first is to consider fourth-order exception operators of types such as

$$(A_{raise} \Rightarrow A_{handle} \Rightarrow [\![\tau]\!]) \Rightarrow [\![\tau]\!]$$

where $A_{raise} = 1 \Rightarrow 1$ and $A_{handle} = (1 \Rightarrow [\![\sigma]\!]) \Rightarrow (1 \Rightarrow [\![\sigma]\!]) \Rightarrow [\![\sigma]\!]$. This affords a very general treatment of exceptions allowing us to model complex dynamic scoping phenomena; however, the relevant exception operators are only available in relatively powerful game models such as $\mathcal{G}_{!_3}$. Another option is to restrict attention to a somewhat more disciplined class of uses of exceptions, in which an independent *raise* operator is eschewed and instead the first argument to *handle* explicitly incorporates the relevant invocations of *raise*: consider *e.g.* $A'_{handle} = ((1 \Rightarrow 1) \Rightarrow [\![\sigma]\!]) \Rightarrow (1 \Rightarrow [\![\sigma]\!]) \Rightarrow [\![\sigma]\!]$. This conveniently accounts for those uses of exceptions that can be reasonably modelled *e.g.* in $\mathcal{G}_{!_1}$, and also gives us the rare pleasure of finding a use for a fifth-order operator!

An important difference arises when one considers local store. Whilst a sensible notion "store cell operator" may be formulated, it turns out that non-trivial

coherent families of such operators never exist. The issue here is that programs of non-ground type involving local store can define functions with *persistent* internal state, which may behave differently each time they are called, and one cannot hope to model such a thing in a λ_c-model. However, it turns out that one can do better on this front by working in a *linear* rather than an intuitionistic framework, as is often done in game semantics to account for stateful behaviour (see *e.g.* [32]).

We also expect a similar treatment of continuations to be possible using suitable higher order operators. However, our approach seems to have very little to offer in the case of non-determinism or input/output, since there is no evident specification for the relevant operators in these cases.

Once some further instances of our approach have been worked out, a detailed comparison of the merits and demerits of the monadic (or algebraic theory) and intensional approaches will be possible. There is, however, one suggestion we would like to make at this stage. In the monadic approach, each effect featuring in a complex language typically requires a separate increment to the model construction. Moreover, if *local* state is to be treated, maybe functor categories must also be added to the mix; for name generation, perhaps nominal sets are required too. By contrast, in the intensional approach, one may be able to account for all these effects using operators that are naturally to hand in a single model — indeed, it seems that there *are* fairly simple models (such as $\mathcal{G}_{!_3}$, or more accurately its linear counterpart) that support virtually all the effect operators one might hope for. If this is so, we regard it as an important point in favour of intensional semantics.

Finally, we note that there are some tantalizing resemblances between our approach and the mechanism employed in Haskell for localization of effects using the `runST` operator [15, 23]. It would be interesting to explore this connection.

References

1. Abramsky, S.: Semantics of interaction: an introduction to game semantics. In A.M. Pitts and P.Dybjer, eds., Semantics and Logics of Computation, CUP (1997) 1-31
2. Abramsky, S., Ghica, D., Murawski, A., Ong, C.-H., Stark, I.: Nominal games and full abstraction for the nu-Calculus. Proc. 19th LICS, IEEE Press (2004) 150–159
3. Abramsky, S., Honda, K., McCusker, G.: A fully abstract game semantics for general references. Proc. 13th LICS, IEEE Press (1998) 334–344
4. Abramsky, S., McCusker, G.: Call-by-value games. Proc. 11th CSL, Springer LNCS **1414** (1998) 1-17
5. Abramsky, S., McCusker, G.: Game semantics. Proc. 1997 Marktoberdorf Summer School, Springer (1999) 1–56
6. Abramsky, S., McCusker, G.: Full Abstraction for Idealized Algol with passive expressions. Theor. Comp. Sci. **227** (1999) 3-42
7. Amadio, R.M., Curien, P.-L.: Domains and Lambda-Calculi, CUP (1998)
8. Berry, G., Curien, P.-L.: Sequential algorithms on concrete data structures. Theor. Comp. Sci. **20** (1982) 265-321

9. Bucciarelli, A., Ehrhard, T., Manzonetto, G.: Not enough points is enough. Proc. 16th CSL, Springer LNCS **4646** (2007) 298-312
10. Curien, P.-L.: On the symmetry of sequentiality. Proc. 9th MFPS, Springer LNCS **802** (1993) 29-71
11. Curien, P.-L.: Notes on game semantics. From the author's web page (2006)
12. Hyland, J.M.E.: Game semantics. In A.M. Pitts and P.Dybjer, eds., Semantics and Logics of Computation, CUP (1997) 131-194
13. Laird, J.: A game semantics of local names and good variables. Proc. FoSSaCS'04, Springer LNCS **2987** (2004)
14. Lambek, J., Scott, P.: Introduction to Higher-Order Categorical Logic, CUP (1986)
15. Launchbury, J., Peyton Jones, S.: State in Haskell. Lisp Symb. Comp. **8** (1995) 193-341
16. Longley, J.R.: Notions of computability at higher types I. Proc. Logic Colloquium 2000, Lecture Notes in Logic, ASL **200** (2005) 32–142
17. Longley, J.R.: On the ubiquity of certain total type structures. Math. Struct. in Comp. Science **17** (2007) 841–953
18. Longley, J.R., Wolverson, N.: Eriskay: a programming language based on game semantics. To be presented at GaLoP III, Budapest (2008)
19. Melliès, P.-A.: Comparing hierarchies of types in models of linear logic. Inf. Comp. **189(2)** (2004) 202-234
20. Moggi, E.: Computational lambda-calculus and monads. LFCS report ECS-LFCS-88-66, University of Edinburgh (1988). A shorter version appeared in Proc. 4th LICS, IEEE Press (1989) 14-23
21. Moggi, E.: An abstract view of programming languages. LFCS report ECS-LFCS-90-113, University of Edinburgh (1989)
22. Moggi, E.: Notions of computation and monads. Inf. and Comp. **93** (1991) 55-92
23. Moggi, E., Sabry, A.: Monadic encapsulation of effects: a revised approach. J. Funct. Prog. **11(6)** (2001) 591-627
24. Peyton Jones, S., Wadler, P.: Imperative functional programming. Proc. 20th POPL, ACM Press (1993)
25. Pitts, A., Stark, I.: Observable Properties of Higher Order Functions that Dynamically Create Local Names, or: What's *new*? Proc. MFCS '93, Springer LNCS **711** (1993) 122-141
26. Plotkin, G.D.: LCF considered as a programming language. Theor. Comp. Sci. **5** (1977) 223-255
27. Plotkin, G.D., Power, A.J.: Algebraic operations and generic effects. Appl. Categ. Struct. **11(1)** (2003) 69-94
28. Shinwell, M.R., Pitts, A.M.: On a monadic semantics for freshness. Theor. Comp. Sci. **342** (2005) 28–55
29. Stark, I.: Free-algebra models for the π-calculus. Theor. Comp. Sci. **390(2-3)** (2008) 248-270
30. Troelstra, A.S. (editor): Metamathematical Investigation of Intuitionistic Arithmetic and Analysis. Springer LNM **344** (1973)
31. Tzevelekos, N.: Full abstraction for nominal general references. Proc. 22nd LICS, IEEE Press (2007) 399-410
32. Wolverson, N.: Game semantics for object-oriented languages. PhD thesis, University of Edinburgh, submitted (2007)